Biblioteka STM32 Standard Peripheral Library, krytycznym okiem

Rosnąca złożoność mikrokontrolerów jednoukładowych powoduje, że obsługa wewnętrznych układów peryferyjnych staje się coraz trudniejsza. Zazwyczaj producenci wychodząc naprzeciw użytkownikom dostarczają darmowe biblioteki programistyczne ułatwiające obsługę układów peryferyjnych. Podobnie jest i w przypadku mikrokontrolerów z rodziny STM32. Układy te należą do dość udanych konstrukcji sprzętowych, czego nie można powiedzieć o udostępnionych bibliotekach.

Najważniejszą biblioteką dostarczaną przez producenta jest *STM32 Standard Peripheral Library*, która zawiera API przeznaczone do obsługi podstawowych bloków peryferyjnych. Dodatkowo, oprócz biblioteki podstawowej dostarczane są inne biblioteki rozwiązujące bardziej zaawansowane zagadnienia. Są to między innymi:

- *SPEEX library* – zawierająca implementację kodeka audio SPEEX
- *Motor control library* – implementującą sterowanie silnikami
- USB device library – zawierającą podstawową obsługę interfejsu USB
- DSP library – zawierająca obsługę podstawowych operacji DSP

Niestety, biblioteka standardowa STM32 ma szereg wad, zdaniem autora dyskryminujących ją w większości zastosowań. Z tego powodu, w wielu projektach możemy zaobserwować częste zjawisko użycia jedynie plików nagłówkowych zawierających deklaracje rejestrów oraz bezpośrednie programowanie układów peryferyjnych z poziomu aplikacji. W niniejszym artykule przedstawimy sposób, w jaki biblioteka może zostać ulepszona poprzez zastosowanie kilku nieskomplikowanych „sztuczek". W wyniku tych zabiegów powstała nowa biblioteka **libstm32**, która jest częścią systemu *ISIX RTOS*. Bazuje ona na bibliotece standardowej producenta, jednak eliminuje większość jej wad, o czym dalej.

Wady biblioteki dostarczanej przez producenta

Śmiało można stwierdzić ze biblioteka standardowa „*STM32 standard peripheral library*" jest sztandarowym przykładem złego projektowania, przy czym do najważniejszych wad biblioteki należą (uszeregowane zgodnie z wagą problemu):

- Duży rozmiar kodu wynikowego, niewspółmierny do funkcjonalności realizowanej przez bibliotekę.
- Przekazywanie parametrów do funkcji poprzez struktury.
- Chaos, a co z tym związane, niepotrzebna duplikacja kodu
- Nadmierne użycie funkcjonalności preprocesora (makra #define), co prowadzi do zaśmiecenia globalnej przestrzeni nazw w C++.

Błędne przykłady zawierające masę zmiennych globalnych, czy np. umieszczenie wszystkich funkcji obsługi przerwań w jednej jednostce kompilacji.

W świetle wprowadzenia na rynek układów STM32F100 przeznaczonych dla segmentu *low-end* oraz zapowiedzi wprowadzenia układów z rdzeniem Cortex-M0 – STM32F0, najistotniejszą wadą biblioteki jest nadmierna wielkość kodu wynikowego, który po dołączeniu do aplikacji powoduje znaczne jej „spuchnięcie". Na przykład biblioteka skompilowana GCC w wersji 4.6.2, zajmuje ponad 10 kB pamięci Flash. O ile dla większych mikrokontrolerów np. z rodziny *Connectivity Line*, ta wielkość kodu dodana do programu jest do zaakceptowania, o tyle w przypadku mikrokontrolerów rodziny *Value Line* kod biblioteki może zajmować nawet 60% pojemności pamięci Flash najmniejszych układów (16 kB). W takich okolicznościach użycie biblioteki mija się z celem, ponieważ po jej dołączeniu zostaje niewiele miejsca na program użytkownika, co może prowadzić do tego, że będziemy zmuszeni do użycia „większego" mikrokontrolera.

Najprostszym sposobem na rozwiązanie problemu rozmiaru biblioteki bez konieczności dokonywania drastycznych zmian jest skompilowanie jej z włączonymi flagami GCC -ffunction-sections oraz -fdata-sections, a następnie użycie flagi linkera --gc-sections. Powoduje to umieszczenie każdej funkcji w odrębnej sekcji, natomiast dodatkowa flaga dla linkera powoduje niedołączanie do kodu wynikowego sekcji, które nie są używane. Dzięki temu będą dołączone tylko te funkcje, które zostały użyte w programie. Rozwiązanie choć stosunkowo proste, nadal jest tylko półśrodkiem. Jeśli korzystamy z dużej liczby funkcji taka optymalizacja jest nie wystarczająca, dalszy sposób optymalizacji będzie wymagał przepisania oraz modyfikacji fragmentów biblioteki.

Kolejnym problemem związanym z wielkością pamięci Flash jest przyjęty model przekazywania większości parametrów funkcji za pomocą pojedynczej struktury, w której zdefiniowano wiele pól konfiguracyjnych. Zapewne zamierzeniem autorów było, zwiększenie czytelności kodu. Niemniej jednak, umieszczenie funkcji biblioteki w odrębnych modułach kompilacji oraz taki sposób przekazywania parametrów powoduje znaczący powiększenie kodu programu. Na przykład nieskomplikowana inicjalizacja układu czasowo licznikowego TIM1, wygląda tak:

```
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
  TIM_TimeBaseStructure.TIM_Period = MAX_PWM_VALUE;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_
CKD_DIV1;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
```

Całość wygląda nieskomplikowanie i przejrzyście, jednak prowadzi do wygenerowania obszernego kodu wynikowego:

```
  //Configure timebase
  /* Time Base configuration */
  TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
```

```
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
  TIM_TimeBaseStructure.TIM_Period = MAX_PWM_VALUE;
 8001b80:       f44f 7200       mov.w   r2, #512
; 0x200
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_
CKD_DIV1;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
 8001b84:       4628            mov     r0, r5
 8001b86:       a908            add     r1, sp, #32
        //Configure timebase
      /* Time Base configuration */
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
  TIM_TimeBaseStructure.TIM_Period = MAX_PWM_VALUE;
 8001b88:       f8ad 2024       strh.w  r2, [sp,
#36]   ; 0x24
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
  TIM_OCInitTypeDef  TIM_OCInitStructure;
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
  TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Disable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_
OutputNState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 256;
 8001b8c:       f44f 7780       mov.w   r7, #256
; 0x100
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_
CKD_DIV1;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
  TIM_OCInitTypeDef  TIM_OCInitStructure;
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
 8001b90:       f04f 0a70       mov.w   sl, #112
; 0x70
  TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Disable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_
OutputNState_Enable;
 8001b94:       f04f 0904       mov.w   r9, #4
  TIM_OCInitStructure.TIM_Pulse = 256;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_Low;
 8001b98:       f04f 0802       mov.w   r8, #2
        //Configure timer in pwm mode
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_TIM1,
ENABLE );
        //Configure timebase
      /* Time Base configuration */
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
 8001b9c:       f8ad 4020       strh.w  r4, [sp,
#32]
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
 8001ba0:       f8ad 4022       strh.w  r4, [sp,
#34]   ; 0x22
  TIM_TimeBaseStructure.TIM_Period = MAX_PWM_VALUE;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_
CKD_DIV1;
 8001ba4:       f8ad 4026       strh.w  r4, [sp,
#38]   ; 0x26
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
 8001ba8:       f88d 4028       strb.w  r4, [sp,
#40]   ; 0x28
  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
 8001bac:       f7ff f886       bl      8000cbc
<TIM_TimeBaseInit>
```

Umieszczenie funkcji bibliotecznych w niezależnych modułach kompilacji powoduje, że każdy parametr zostaje odłożony na stosie, po czym zostaje wywołana właściwa funkcja, a następnie w samej funkcji parametry odczytywane są ponownie ze stosu i wykonywane są na nich żądane operacje. Jak łatwo zauważyć, w tym przypadku do funkcji są przekazywane wartości stałe. Kompilator jednak nie ma możliwości optymalizacji kodu między modułami i funkcja *TIM_TimeBaseInit*, musi być skompilowana, tak jakby wszystkie parametry były zmiennymi. Spójrzmy zatem na wynik działania kompilatora dla tej funkcji:

```
  if((TIMx == TIM1) || (TIMx == TIM8)|| (TIMx ==
TIM2) || (TIMx == TIM3)||
 8000ce6:       b983            cbnz    r3, 8000d0a
<TIM_TimeBaseInit+0x4e>
 8000ce8:       f1a0 4c80       sub.w   ip, r0,
#1073741824     ; 0x40000000
 8000cec:       f1dc 0500       rsbs    r5, ip, #0
 8000cf0:       f44f 6280       mov.w   r2, #1024
; 0x400
 8000cf4:       f2c4 0200       movt    r2, #16384
; 0x4000
 8000cf8:       eb45 050c       adc.w   r5, r5, ip
 8000cfc:       4290            cmp     r0, r2
 8000cfe:       bf14            ite     ne
 8000d00:       462a            movne   r2, r5
 8000d02:       f045 0201       orreq.w r2, r5, #1
 8000d06:       2a00            cmp     r2, #0
 8000d08:       d043            beq.n   8000d92
<TIM_TimeBaseInit+0xd6>
     (TIMx == TIM4) || (TIMx == TIM5))
  {
    /* Select the Counter Mode */
    tmpcr1 &= (uint16_t)(~((uint16_t)(TIM_CR1_DIR |
TIM_CR1_CMS)));
 8000d0a:       f64f 728f       movw    r2, #65423
; 0xff8f
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct-
>TIM_CounterMode;
 8000d0e:       884d            ldrh    r5, [r1, #2]

  if((TIMx == TIM1) || (TIMx == TIM8)|| (TIMx ==
TIM2) || (TIMx == TIM3)||
     (TIMx == TIM4) || (TIMx == TIM5))
  {
    /* Select the Counter Mode */
    tmpcr1 &= (uint16_t)(~((uint16_t)(TIM_CR1_DIR |
TIM_CR1_CMS)));
 8000d10:       4022            ands    r2, r4
    tmpcr1 |= (uint32_t)TIM_TimeBaseInitStruct-
>TIM_CounterMode;
 8000d12:       ea42 0405       orr.w   r4, r2, r5
  }
```

Pomimo tego, że jako pierwszy argument przekazano wartość stałą TIM1, w kodzie funkcji następuje sprawdzenie warunków pierwszego parametru, tak jakby to była wartość zmienna. Dzieje się tak, ponieważ kompilator domyślnie nie ma możliwości optymalizowania kodu pomiędzy poszczególnymi modułami kompilacji. O ile w przypadku wielokrotnego wywoływania funkcji z różnymi parametrami taki model sprawdza się, o tyle w przypadku pojedynczych wywołań funkcji ze stałymi parametrami nie jest optymalny.

Zauważmy, że zazwyczaj konfigurując układy peryferyjne robimy to jeden raz po uruchomieniu i zwykle używamy do tego stałych wartości. Na przykład na początku programu konfigurujemy układ TIM1, tak aby zliczał do 100 i nigdzie więcej w programie nie potrzebujemy robić tego ponownie. Niestety, z powodu braku możliwości optymalizacji do aplikacji zostanie dołączone dużo nadmiarowego kodu z biblioteki, który nigdy nie będzie użyty. O tym w jaki sposób zoptymalizować kod biblioteki standardowej za pomocą funkcji inline dowiemy się z dalszej części artykułu.

Innym problemem prowadzącym do duplikacji kodu jest, nadmierny chaos, który najprawdopodobniej wynika z błędnych założeń projektowych (lub ich braku). Weźmy np. rodzinę funkcji konfigurujących bloki CC (*Compare-Capture*) w trybie porównania układów czasowo-licznikowych. W zależności od wybranego układu czasowo-licznikowego możemy mieć do 4 układów CC, które inicjalizowane są za pomocą 4 oddzielnych funkcji:

```
void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_
OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_
OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC3Init(TIM_TypeDef* TIMx, TIM_
OCInitTypeDef* TIM_OCInitStruct);
void TIM_OC4Init(TIM_TypeDef* TIMx, TIM_
OCInitTypeDef* TIM_OCInitStruct);
```

Zupełnie nielogiczne jest to, że funkcje jako parametr przyjmują wskaźnik do struktury układu czasowo-licznikowego, natomiast, numer układu *Compare-Capture* jest rozróżniany na podstawie wywołania odpowiedniej funkcji. Jeżeli teraz spojrzymy na kod możemy zauważyć, że każda z tych funkcji wykonuje te same czynności, zmieniając jedynie zestawy rejestrów oraz przesunięcia bitowe. Jak łatwo zauważyć studiując noty katalogowe mikrokontrolera, bity konfiguracyjne układów CC układają się w logiczną całość i nie ma potrzeby tworzenia dla nich oddzielnych funkcji. Prowadzi to tylko do zbędnej duplikacji kodu oraz stwarza dodatkową możliwość popełnienia błędów. Jest to klasyczny przykład antywzorca projektowego „*Copy and paste programming*". W bibliotece możemy znaleźć wiele podobnych przypadków, występowania tego antywzorca.

Kolejnym problemem z wykorzystaniem biblioteki, w kontekście języka C++ jest nadmierne wykorzystanie makrodefinicji czyli preprocesora języka C, głównie do definiowania stałych przez #define. Jedynym wsparciem związanym z C++ jest występujące w nagłówkach extern C {}. Ponieważ preprocesor dokonuje translacji pliku przed kompilacją, definiowanie stałych przez #define, powoduje, że znajdują się one w globalnej przestrzeni nazw, co prowadzi do jej zaśmiecenia. Również na etapie debugowania, używanie dużej ilości makr jest kłopotliwe, ponieważ debugger nie ma możliwości ich śledzenia.

Ostatnim problemem są przykłady ilustrujące użycie biblioteki, które raczej powinny zostać nazwane „antyprzykładami". Są one napisane w sposób bardzo chaotyczny, zawierają szereg przykładów programowania przez zmienne globalne, który stanowi antywzorzec zwany „*enemy of encapsulation*". Dodatkowo, sprzyja temu przyjęty w bibliotece model obsługi przerwań które umieszczone są w jednym module stanowiącym globalny worek na wszystkie funkcje obsługi przerwań (plik *stm32_it.c*). Aby nie być gołosłownym, spójrzmy na początek przykładu użycia biblioteki do obsługi interfejsu I²S:

```
I2S_InitTypeDef I2S_InitStructure;
const uint16_t I2S3_Buffer_Tx[32] =
 {0x0102, 0x0304, 0x0506, 0x0708, 0x090A, 0x0B0C,
  0x0D0E, 0x0F10, 0x1112, 0x1314, 0x1516, 0x1718,
  0x191A, 0x1B1C, 0x1D1E, 0x1F20, 0x2122, 0x2324,
  0x2526, 0x2728, 0x292A, 0x2B2C, 0x2D2E, 0x2F30,
  0x3132, 0x3334, 0x3536, 0x3738, 0x393A, 0x3B3C,
  0x3D3E, 0x3F40};
uint16_t I2S2_Buffer_Rx[32];
__IO uint32_t TxIdx = 0, RxIdx = 0;
TestStatus TransferStatus1 = FAILED, TransferStatus2
= FAILED;
ErrorStatus HSEStartUpStatus;
```

Widzimy tutaj szereg zmiennych globalnych, chociażby struktura wykorzystana do inicjalizacji I²S I2S_InitStructure, która jest wykorzystana tylko w jednej funkcji i powinna być zdefiniowana jako zmienna automatyczna. Pozostałe zmienne globalne, co stanowi znacznie gorszy antywzorzec wykorzystywane są przez inny moduł, w którym zdefiniowane są wszelkie funkcje obsługi przerwań (*stm32_it.c*):

```
void SPI3_IRQHandler(void)
{
  /* Check the interrupt source */
  if (SPI_I2S_GetITStatus(SPI3, SPI_I2S_IT_TXE) ==
SET)
  {
    /* Send a data from I2S3 */
    SPI_I2S_SendData(SPI3, I2S3_Buffer_Tx[TxIdx++]);
  }

  /* Check the end of buffer transfer */
  if (RxIdx == 32)
  {
    /* Disable the I2S3 TXE interrupt to end the
communication */
    SPI_I2S_ITConfig(SPI3, SPI_I2S_IT_TXE, DISABLE);
  }
}

/**
  * @brief  This function handles SPI2 global
interrupt request.
  * @param  None
  * @retval None
  */
void SPI2_IRQHandler(void)
{
  /* Check the interrupt source */
  if (SPI_I2S_GetITStatus(SPI2, SPI_I2S_IT_RXNE) ==
SET)
  {
    /* Store the I2S2 received data in the relative
data table */
    I2S2_Buffer_Rx[RxIdx++] = SPI_I2S_
ReceiveData(SPI2);
  }
}
```

Zmienne globalne dostępne są z każdego miejsca w programie, i mogą zostać zmienione przez dowolne moduły w dowolnym miejscu, co w momencie przypadkowego nadpisania może prowadzić do wielu nieoczekiwanych skutków, oraz może spowodować duże trudności w lokalizacji problemu.

Trudno tutaj, o przykład gorszego stylu programowania, a wszystko to przecież znajduje się w przykładach, które powinny mieć charakter edukacyjny, i powinny skupiać się nie tylko na samych peryferiach, ale również na prawidłowych wzorcach, tak aby korzystający nie nabywał od samego początku złych nawyków.

newlibstm32 – Nowa biblioteka bazująca na bibliotece oryginalnej

Wspomniane wyżej wady skłoniły autora do opracowania nowej biblioteki obsługi układów peryferyjnych, w maksymalnym stopniu bazujące na oryginalnej bibliotece od ST, ale pozbawione wyżej wspomnianych wad. W ten sposób powstała biblioteka newlib-

stm32, która stanowi część systemu ISIXRTOS. Nic nie stoi jednak na przeszkodzie, aby używać jej zupełnie niezależnie w projekcie bez systemu operacyjnego. Najbardziej aktualną wersje biblioteki możemy zawsze pobrać za z pomocą systemu kontroli wersji Mercurial (*http://mercurial.selenic.com/*), spod adresu *http://www.boff.pl/hg/isix/isixrtos/*. Wydając polecenie *hg clone http://www.boff.pl/hg/isix/isixrtos* lub wykorzystując opcję *clone* graficznego narzędzia TortoiseHg. (**rysunek 1**).

Biblioteka stara się w jak największym stopniu nawiązywać do nazw funkcji użytych w oryginalnej bibliotece, tak aby stosunkowo niewielkim nakładem kosztów można było zmodyfikować programy używające oryginalnych bibliotek. Wszystkie funkcje, które w starej bibliotece używały nomenklatury *PERIPH_FunctionName* w nowej bibliotece mają nazwy p*eriph_function_name*. Dodatkowym założeniem w stosunku do biblioteki oryginalnej jest to, że w języku C++ wszystkie funkcje biblioteczne dostępne są w przestrzeni nazw stm32.

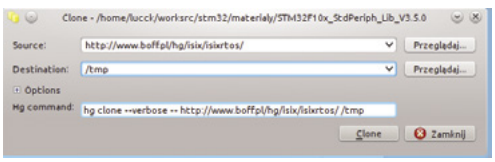
Spójrzmy teraz w jaki sposób dano sobie radę z opisywanymi wcześniej wadami biblioteki oryginalnej. Duża wielkość kodu została ograniczona dzięki zastosowaniu funkcji inline. Definicje wszystkich funkcji bibliotecznych umieszczone są w plikach nagłówkowych, a każda funkcja zadeklarowana jest jako funkcja inline. Dzięki temu, kompilator ma możliwość przeprowadzenia dokładnej optymalizacji kodu w momencie wywołania. Dodatkowo, tylko używane funkcje są kompilowane i zajmują dodatkowy obszar pamięci.

Aby umożliwić dalszą optymalizację, zmieniono sposób przekazywania parametrów do funkcji, które zamiast struktur używają większej liczby argumentów. Rozwiązanie to jest może odrobinę mniej czytelne, ale przy wykorzystaniu współczesnych, podpowiadających składnię edytorów nie ma to znaczenia. Spójrzmy teraz jak wygląda fragment inicjalizacji układu czasowo-licznikowego TIM1 z użyciem nowej biblioteki:

```
//Configure timebase
  stm32::tim_timebase_init( TIM1, 0, TIM_
CounterMode_Up, MAX_PWM_VALUE, TIM_CKD_DIV1, 0 );
, a teraz zobaczmy jaki kod maszynowy powstał
w wyniku użycia powyższej funkcji:

static inline void tim_timebase_init(TIM_TypeDef*
TIMx, uint16_t prescaler, uint16_t cnt_mode,
uint16_t period, uint16_t clkdiv, uint8_t
rptcounter)
{
  uint16_t tmpcr1 = 0;
  tmpcr1 = TIMx->CR1;
 8000368:  f44f 5330   mov.w   r3, #11264      ;
0x2c00
 800036c:  f440 6000   orr.w   r0, r0, #2048   ;
0x800
 8000370:  f2c4 0301   movt    r3, #16385      ;
0x4001
 8000374:  6188        str     r0, [r1, #24]
 8000376:  8819        ldrh    r1, [r3, #0]
  /* Set the Autoreload value */
  TIMx->ARR = period ;
  /* Set the Prescaler value */
  TIMx->PSC = prescaler;
  8000378:   2500        movs    r5, #0
}

if((TIMx != TIM6) && (TIMx != TIM7))
{
  /* Set the clock division */
  tmpcr1 &= (uint16_t)(~((uint16_t)TIM_CR1_CKD));
 800037a:    f421 715c bic.w   r1, r1, #880    ;
0x370
 800037e:    0409      lsls    r1, r1, #16
 8000380:    0c09      lsrs    r1, r1, #16
  tmpcr1 |= (uint32_t)clkdiv;
}
  TIMx->CR1 = tmpcr1;
 8000382:    8019      strh    r1, [r3, #0]
  /* Set the Autoreload value */
  TIMx->ARR = period ;
 8000384:    f44f 7000 mov.w   r0, #512        ;
0x200
  TIMx->RCR = rptcounter;
}

  /* Generate an update event to reload the
Prescaler and the Repetition counter values
immediately */
TIMx->EGR = TIM_PSCReloadMode_Immediate;
 8000388:    2101      movs    r1, #1
}
  TIMx->CR1 = tmpcr1;
  /* Set the Autoreload value */
  TIMx->ARR = period ;
 800038a:    8598      strh    r0, [r3, #44]   ;
0x2c
  /* Set the Prescaler value */
  TIMx->PSC = prescaler;
 800038c:    851d      strh    r5, [r3, #40]   ;
0x28
  if ((TIMx == TIM1) || (TIMx == TIM8)|| (TIMx ==
TIM15)|| (TIMx == TIM16) || (TIMx == TIM17))
  {
    /* Set the Repetition Counter value */
    TIMx->RCR = rptcounter;
 800038e:    861d      strh    r5, [r3, #48]   ;
0x30
}
```

Jak możemy zauważyć cały rozmiar kodu wynikowego funkcji, w nowej biblitotece, zajmuje podobny obszar pamięci FLASH, co w starej biblitotece zajmowało przekazanie parametrów, nie licząc kodu który zajmowała sama funkcja. Kompilator rozwijając funkcję inline dokonał znaczącej optymalizacji kodu, który w zasadzie sprowadza się do bezpośredniego przepisania przekazanych wartości, do rejestrów mikrokontrolera. W tym przypadku zyskujemy zarówno dużo mniejszą zajętość pamięci FLASH, jak i również większą szybkość wykonania kodu (atrybut w zasadzie bez znaczenia przy funkcjach inicjalizujących). Podobnie w przypadku rodzin funkcji odpowiedzialnych za odczyt statusu flag np. `static inline bool tim_get_flag_status(TIM_TypeDef* TIMx, uint16_t TIM_FLAG)` kompilator optymalizuje funkcje tak aby sprowadzała się do bezpośredniego odczytania rejestru układu peryferyjnego, co prowadzi do użycia pojedynczych instrukcji maszynowych.

Niepotrzebna duplikacja kodu w funkcjach, które w zasadzie robią to samo, ale operują na różnych danych została rozwiązana poprzez napisanie duplikujących się funkcji od nowa i zdefiniowanie dodatkowych parametrów. Na przykład wspomniane wcześniej funkcje `void TIM_OC1Init, void TIM_OC2Init, void TIM_OC3Init, void TIM_OC4Init` zostały zamienione na poje-



Rysunek 1. Okno programu TortoiseHg, opcja clone

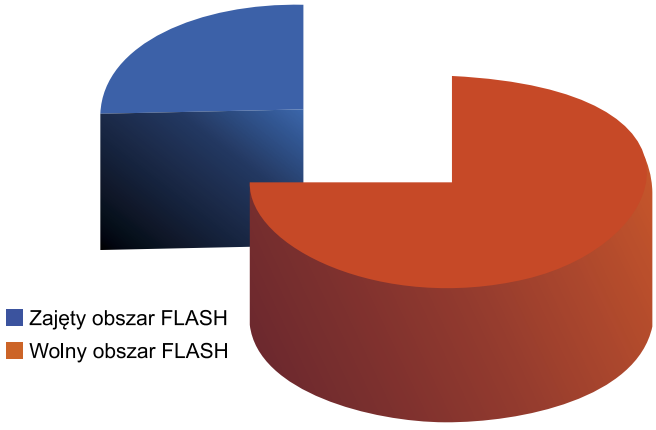
dynczą funkcję `static inline void tim_oc_init(TIM_TypeDef* TIMx, enum tim_cc_chns chn, uint16_t oc_mode, uint16_t oc_value, uint16_t output_state, uint16_t output_n_state, uint16_t polarity,uint16_t n_polarity, uint16_t idle_state, uint16_t n_idle_state)`, do której w dodatkowym parametrze możemy przekazać numer bloku CC o który nam chodzi. Zmiana ta zapewnia dobrą optymalizację, ponieważ jest to funkcja *inline*, ale również dodatkowo, likwidowane są potencjalne źródła innych problemów wynikających z powielania kodu przez kopiowanie.

Kolejny problem z poprawieniem funkcjonalności składniowej biblioteki dla języka C++ rozwiązano poprzez zmianę definicji stałych #define na wartości stałe static const , w przypadku, gdy kompilowany jest moduł języka C++. Stałe definiowane są po staremu za pomocą #define jedynie, gdy kompilowany jest moduł języka C. Dzięki temu stałe w języku C++ dostępne są w przestrzeni nazw stm32, zamiast w przestrzeni globalnej.

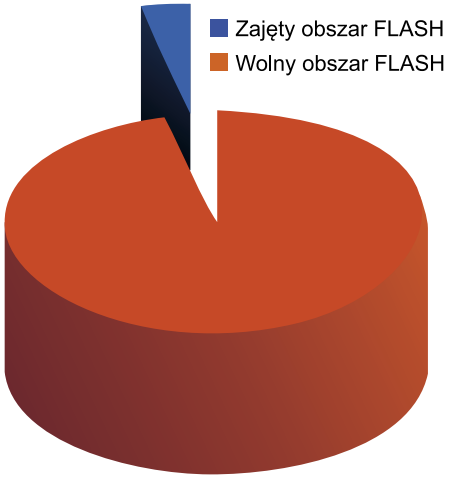
Przykład praktyczny

Porównajmy teraz obie wersje biblioteki, pod względem zajmowanych zasobów na przykładzie, którego zadaniem jest inicjalizacja mikrokontrolera, tak aby na liniach *PA8* oraz *PB13*, wygenerować zależne sygnały PWM odwrócone w fazie, a na linii *PB14*, wygenerować dodatkowo niezależny sygnał PWM. Kod zostanie przygotowany dla mikrokontrolera STM32F100R6T6B, posiadającego 32kB pamięci FLASH Sygnały PWM zostaną wygenerowane za pomocą zaawansowanego układu czasowo-licznikowe*go TIM1*. Kod programu realizującego to zadanie z wykorzystaniem oryginalnej biblioteki wygląda następująco:

```
int pwm_setup()
{

    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = XNEG_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_
Speed_50MHz;
    //Configure xdbl pin To PWM function
        GPIO_Init(XDBL_PORT, &GPIO_InitStructure);
        //Configure timer in pwm mode
    RCC_APB2PeriphClockCmd( RCC_APB2Periph_TIM1,
ENABLE );
        //Configure timebase
      /* Time Base configuration */
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
  TIM_TimeBaseStructure.TIM_Prescaler = 0;
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_
CounterMode_Up;
  TIM_TimeBaseStructure.TIM_Period = MAX_PWM_VALUE;
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_
CKD_DIV1;
  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;

  TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
  TIM_OCInitTypeDef  TIM_OCInitStructure;
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
  TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Disable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_
OutputNState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 256;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_Low;
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_
OCNPolarity_Low;
  TIM_OCInitStructure.TIM_OCIdleState = TIM_
OCIdleState_Reset;
  TIM_OCInitStructure.TIM_OCNIdleState = TIM_
OCNIdleState_Reset;
  TIM_OC2Init(TIM1, &TIM_OCInitStructure);

    //Configure OC1/IC1N output XPOS, XNEG PIN
  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2;
  TIM_OCInitStructure.TIM_OutputState = TIM_
OutputState_Enable;
  TIM_OCInitStructure.TIM_OutputNState = TIM_
OutputNState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 256;
  TIM_OCInitStructure.TIM_OCPolarity = TIM_
OCPolarity_Low;
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_
OCNPolarity_High;
  TIM_OCInitStructure.TIM_OCIdleState = TIM_
OCIdleState_Reset;
  TIM_OCInitStructure.TIM_OCNIdleState = TIM_
OCNIdleState_Reset;
  TIM_OC1Init(TIM1, &TIM_OCInitStructure);

    //Enable preload register
    TIM_OC1PreloadConfig(TIM1, ENABLE);

    TIM_BDTRInitTypeDef TIM_BDTRInitStructure;

    /* Automatic Output enable, Break, dead time
and lock configuration*/
  TIM_BDTRInitStructure.TIM_OSSRState = TIM_
OSSRState_Disable;
  TIM_BDTRInitStructure.TIM_OSSIState = TIM_
OSSIState_Disable;
  TIM_BDTRInitStructure.TIM_LOCKLevel = TIM_
LOCKLevel_1;
  TIM_BDTRInitStructure.TIM_DeadTime = PWM_DEAD_
TIME;
  TIM_BDTRInitStructure.TIM_Break = TIM_Break_
Disable;
  TIM_BDTRInitStructure.TIM_BreakPolarity = TIM_
BreakPolarity_Low;
  TIM_BDTRInitStructure.TIM_AutomaticOutput = TIM_
AutomaticOutput_Enable;

  TIM_BDTRConfig(TIM1, &TIM_BDTRInitStructure);
```



Rysunek 2. Wielkość kodu w pamięci Flash przed zmodyfikowaniem biblioteki wynosi aż 11472 bajtów, co stanowi aż 35% pojemności pamięci Flash



Rysunek 3. Kod otrzymany w wyniku użycia zmodyfikowanej biblioteki zajmuje 1292 bajty tj. blisko 10 razy mniej

```
        //Enable outputs and timer

  /* Main Output Enable */
  TIM_CtrlPWMOutputs(TIM1, ENABLE);
  TIM_Cmd(TIM1, ENABLE);
}
```

Widzimy że kod inicjalizujący jest stosunkowo skomplikowany, oraz dodatkowo wymaga użycia kilku struktur, zadeklarowanych jako zmienne lokalne. Do przykładu z kodu biblioteki STM32STDLIB zostały włączone jedynie moduły obsługi timera, bloku RCC, oraz portów GPIO, a pozostałe moduły nie są w ogóle kompilowane. Pomimo tych zabiegów wielkość pamięci Flash zajmowanej przez tak prosty przykład wynosi aż 11472 bajtów co stanowi aż 35% wielkości pamięci Flash (**rysunek 2**). Spójrzmy teraz na kod tego samego przykładu napisanego z wykorzystaniem nowej wersji biblioteki:


```
int pwm_setup()
{
  //Configure xdbl pin To PWM function
  stm32::io_config( XDBL_PORT, XDBL_PIN,
stm32::GPIO_MODE_50MHZ, stm32::GPIO_CNF_ALT_PP );
  //Configure timer in pwm mode
  stm32::rcc_periph_clock_cmd( stm32::rcc_clk_apb2,
RCC_APB2Periph_TIM1, true );
  //Configure timebase
  stm32::tim_timebase_init( TIM1, 0, TIM_
CounterMode_Up, MAX_PWM_VALUE, TIM_CKD_DIV1, 0 );
  //Configure OC2N output only XNEG PIN
  stm32::tim_oc_init( TIM1, stm32::tim_cc_chn2,
TIM_OCMode_PWM2, 256,
  TIM_OutputState_Disable, TIM_OutputNState_Enable,
TIM_OCPolarity_Low,
  TIM_OCNPolarity_Low, TIM_OCIdleState_Reset, TIM_
OCNIdleState_Reset );
  //Configure the PWM main output
  //Configure xpos, xneg pin To PWM function
  stm32::io_config( XPOS_PORT, XPOS_PIN,
stm32::GPIO_MODE_50MHZ, stm32::GPIO_CNF_ALT_OD );
  stm32::io_config( XNEG_PORT, XNEG_PIN,
stm32::GPIO_MODE_50MHZ, stm32::GPIO_CNF_ALT_PP );
  //Configure OC1/IC1N output XPOS, XNEG PIN
  stm32::tim_oc_init( TIM1, stm32::tim_cc_chn1,
TIM_OCMode_PWM1, 0,
  TIM_OutputState_Enable, TIM_OutputNState_Enable,
TIM_OCPolarity_Low,
  TIM_OCNPolarity_High, TIM_OCIdleState_Reset, TIM_
OCNIdleState_Reset );
  //Enable preload register
  stm32::tim_oc_preload_config( TIM1, stm32::tim_cc_
chn1, true );
  //Configure dead time
  stm32::tim_bdtr_config( TIM1, PWM_DEAD_TIME,
TIM_OSSRState_Disable|TIM_OSSIState_Disable |TIM_
LOCKLevel_1
| TIM_Break_Disable | TIM_BreakPolarity_Low
| TIM_AutomaticOutput_Enable );
  //Enable outputs and timer
  stm32::tim_ctrl_pwm_outputs( TIM1, true );
  stm32::tim_cmd( TIM1, true );
}
```

Przykład jest bardziej zwięzły, oraz nie wymaga deklarowania dodatkowych struktur w postaci zmiennych lokalnych. Po skompilowaniu kompletny przykład załadowany do pamięci FLASH zajmuje jedynie 1292 bajty, przy czym cały kod inicjalizujący blok PWM zajmuje jedynie 268 bajtów. Kompletny przykład stanowi zatem niewielką część pamięci FLASH mikrokontrolera (**rysunek 3**). Dzięki użyciu funkcji inline tylko te fragmenty które są używane włączane są do kodu wynikowego.

Podsumowanie

Stosując jedynie proste zmiany, udało się zbudować nową wersję biblioteki, którą możemy używać we własnych aplikacjach, bez obawy o nadmierną zajętość pamięci FLASH. Możemy zatem w pełni korzystać z dobrodziejstwa biblioteki układów peryferyjnych, nawet w najmniejszych układach rodziny STM32, bez konieczności uciekania się do bezpośredniego programowania rejestrów.

Lucjan Bryndza, EP